EECS 370 Midterm Exam

Winter 2021

This exam is presented with modifications.

This exam was originally released through gradescope in a virtual semester. Some comments have been added to reflect modifications for a paper exam.

Question 1: True/false and multiple choice

11 of these were randomly selected.

1. A struct with a nested struct is (always/sometimes/never) aligned with the size of the largest nested struct
2. Symbolic addresses must be resolved during the linking process (True/False)
3. Using big endian or small endian will alter which bytes of memory get loaded into a register (True/False)
4. Single cycle programs (always/sometimes/never) have a lower CPI than a multi cycle program
5. sw instructions (always/sometimes/never) require a read from memory
6. Registers in ARM are byte-addressable (True/False)
7. main() (always/sometimes/never) needs to use a caller-save
8. Increasing the number of registers in LC2K would (definitely would/might/definitely would not) limit the number of possible instructions
9. You can use global labels with a jalr instruction (True/False)
10. CISC processors are (always/sometimes/never) faster than RISC processors
11. A valid LC2K program (one that could successfully execute to completion on your 1s simulator) needs to have a "halt" instruction in its assembly code (True/False)
12. Using callee save is the best way to minimize loads/stores in a program (True/False)
13. It is important to check the contents of a register before doing a callee save (True/False)
14. For an individual save, a caller save (always/sometimes/never) uses less assembly instructions than a callee save
15. IEEE floating point numbers and 2's complement numbers handle negative values in the same way (True/False)
16. In general, a LEGv8 program requires fewer memory accesses than a LC2K program executing the same actions (True/False)
17. It is (always/sometimes/never) possible to compile a valid LC2K program into C
18. Global variables are sometimes stored on the stack while the program is running (True/False)
19. Multi-cycle data paths always have faster hardware than single cycle data paths (True/False)
20. ARM assembly code can be written and run on a 32-bit system (True/False)

21. Multi cycle programs (always/sometimes/never) have a lower CPI than a single cycle program
22. IEEE floating point numbers and 2’s complement numbers handle negative values in different ways (True/False)
23. It is (always/sometimes/never) possible to deconstruct a C program into LC2K
24. Multi-cycle data paths sometimes have faster hardware than single cycle data paths (True/False)
25. Using caller save is the best way to minimize loads/stores in a program (True/False)
26. RISC processors are (always/sometimes/never) faster than CISC processors

Question 2: Memory Alignment (Version A)

(a) Use the code below to fill in the following table for a 64-bit, byte-addressable architecture. Assume that “michigan” begins at address 1000 (in decimal).

```
struct {
     float weiser; 8      1000- 1007
     long ugli;  8        1008- 1015
     char mojo; 1         1016-1016
     struct {                                  1017-1023 (P)
          short 8 *duderstadt;   1024 -1031
          int 4 bursley;         1032- 1035
          char 1 stamps;         1036-1036.
     } north; 1024-1039 (16)                    1037- 1039 (P).
     char 8 *ross[3];
                =   1040 -1047
} michigan;         1048 - 1055
1000- 1063          1056 - 1063
  (64)
```

Variable	Size (bytes)	Start Address (Decimal)	End Address (Decimal)
weiser	8	1000	1007
ugli	8		
mojo	1		
duderstadt	8		
bursley	4		
stamps	1		
north	16		
ross	24		
michigan	64	1000	1063.

Question 2: Memory Alignment (Version B)

(a) Use the code below to fill in the following table for a 64-bit, byte-addressable architecture. Assume that “michigan” begins at address 1000 (in decimal).

```
struct {
      char weiser;
      long ugli;
      float mojo;
      struct {
            short duderstadt;
            char *bursley;
            int stamps;
      } north;
      char *ross[3];
} michigan;
```

Variable	Size (bytes)	Start Address (Decimal)	End Address (Decimal)
weiser			
ugli			
mojo			
duderstadt			
bursley			
stamps			
north			
ross			
michigan			

Question 2: Memory Alignment (Version C)

(a) Use the code below to fill in the following table for a 64-bit, byte-addressable architecture. Assume that “michigan” begins at address 1000 (in decimal).

```
struct {
     float weiser;
     struct {
          short *duderstadt;
          int bursley;
          char stamps;
     } north;
     long ugli;
     char mojo;
     char *ross[3];
} michigan;
```

Variable	Size (bytes)	Start Address (Decimal)	End Address (Decimal)
weiser			
duderstadt			
bursley			
stamps			
north			
ugli			
mojo			
ross			
michigan			

Question 2: Memory Alignment (Version D)

(a) Use the code below to fill in the following table for a 64-bit, byte-addressable architecture. Assume that “michigan” begins at address 1000 (in decimal).

```
struct {
     char weiser;
     struct {
          short duderstadt;
          char *bursley;
          int stamps;
     } north;
     long ugli;
     float mojo;
     char *ross[3];
} michigan;
```

Variable	Size (bytes)	Start Address (Decimal)	End Address (Decimal)
weiser			
duderstadt			
bursley			
stamps			
north			
ugli			
mojo			
ross			
michigan			

Question 2: Memory Alignment (Version E)

(a) Use the code below to fill in the following table for a 64-bit, byte-addressable architecture. Assume that “michigan” begins at address 1000 (in decimal).

```
struct {
     long weiser;
     float ugli;
     char mojo;
     struct {
          short *duderstadt;
          char bursley;
          int stamps;
     } north;
     char *ross[3];
} michigan;
```

Variable	Size (bytes)	Start Address (Decimal)	End Address (Decimal)
weiser			
ugli			
mojo			
duderstadt			
bursley			
stamps			
north			
ross			
michigan			

Question 3: Assembling and Linking

a) Fill in the rest of the symbol and relocation tables for the following two C files. Note that not all entries in the tables below may be used.

eecs370.c

```
typedef struct exam {
  int umid;
  char* answers;
  int score;
} exam_t;

extern void GradeSub(exam_t* sub);
char* key;
extern int overdraft_fee;

void GradeAll(exam_t all[],
    int num_subs) {

  for(int i=0; i < num_subs; ++i) {
    GradeSub(&all[i]);   BL

    if(all[i].score < 0) {
      all[i].score -= overdraft_fee;
    }                       LDUR.
  }
}
```

crabster.c

```
1   #include "string.h"
2   #include "stdlib.h"
3
4   typedef struct exam {
5     int umid;
6     char* answers;
7     int score;
8   } exam_t;
9
10  #define MERCY 80
11  extern char* key;
12
13  void GradeSub(exam_t* sub) {
14    sub->score = 0;
15
16    static int pinch_count = 0;
17    int crab_factor = rand() % 100;
18                      BL
19    //   (\/) (',,,') (\/)
20    if(crab_factor > MERCY) {
21      sub->answers[0] = '\0';
22      sub->score -= crab_factor - 90;
23      pinch_count++; LDUR & StuR.
24    }
25            BL                  LDUR
26    if(!strcmp(sub->answers, key)) {
27      sub->score = 100;
28    }
29  }
```

eecs370.o Symbol Table

Symbol	Type (T/D/U)

crabster.o Symbol Table

Symbol	Type (T/D/U)

eecs370.o Relocation Table

Line	Instruction (LDUR, STUR, BL)	Symbol

crabster.o Relocation Table

Line	Instruction (LDUR, STUR, BL)	Symbol

True or [✓ False]: A linker can successfully produce an executable with only eecs370.c and crabster.c as inputs.

Question 4: Callee and Caller

```
void foo(void) {
    int r1 = 2, r2 = 0, r3 = 0;
    int r4 = 5;
    printf("This is a print statement\n");
    r2 = r1 + r3 + r4;
    bar();

    for(; r2 != 0; r2--) {
        printf("This is a while loop");
    }
    r4 = r3;

    bar();
}

void bar(void) {
    int r5 = 0, r6 = 0;
    for(; r5 < 5; r5++) {
        printf("This is a for loop\n");
    }
}
```

Consider the above C code. Assume all variables are mapped to registers of the same name.

A. If all of the registers were **caller saved**, how many load/store instruction pairs would be **executed** by the program for the variables **r2, r3, r4, r5 and r6** if foo is called once?
<u>Assume the only analysis the compiler can perform is liveness analysis.</u>

Register	**# of Load/Store Instruction Pairs**
r1	**1**
r2	
r3	
r4	
r5	
r6	

B. Now consider the case where all registers are **callee saved**. How many total load/store instruction pairs will be **executed** if foo is called once in this case?

# of Load/Store Pairs Executed:	

C. Minimize the number of load/store instructions that need to be executed by assigning each register to be either caller or callee saved.

Register	**Caller or Callee**
r1	
r2	
r3	
r4	
r5	
r6	

Question 5: C-to-ARM and the SODA370 factory

The SODA370 factory produces soda packs with a multiple of four soda cans in each pack. However, some packing machines are starting to malfunction, making some soda packs contain a number of soda cans not divisible by four. Someone has written a quality check function below in C to check how many soda packs contain incorrect amounts of soda cans. Convert part of this function into ARM.

Assume `X0` contains the starting address of `data[]`. The function **`quality_check`** returns the amount of bad soda packs in register `X2`.

C

```
struct soda_pack {       4
    char pack_id[4]; //4B          ) <<3
    int64_t num_cans; // 8B

};  16.                            12.

int64_t                  X2
quality_check(struct soda_pack data[]) {
    int32_t bad_packs = 0;
    for(int32_t i = 0; i < 256; i++) {      i <- X3
        if(data[i].num_cans % 4 != 0){
            bad_packs++;
        }
    }    X0
    return bad_packs;
}          ↖ X2     ~~0 0 011
```

ARM

```
qcheck:
          MOVI    X2,     #0x00
          MOVI    X3,     #0x00
loop:
          CMPI    X3,     #256.
          B.GE    end.
          LSL     X4,     X3,     ~~#3~~ #4
          ADD     X5,     X4,     X0
          LDUR    X6,     [X5,    ~~#4~~ #8]
   AND ~~LSR~~    X7,     X6,     ~~#2~~ #3
          CMPI    X7,     #0
          B.EQ    good
bad:
          ADDI    X2,     #1
good:
          ADDI    X3,     #1
          B       loop
end:
```

***Other versions compared i with 1204, 255, 127, and 300.**
Some other versions did a modulus with 8 instead of 4 to check packs not divisible by 8.

Question 6: Single Cycle/Multi Cycle/Pipeline Performance (Version A)

Below are the delays for each component considered in the LC2K single and multicycle datapaths discussed in lecture:

Memory Read: 12 ns
Memory Write: ?
Read Register: 6 ns
Write Register: ?
ALU: ?
All Other Operations: 0 ns

Assume the following delays for the instructions given below. Calculate the missing ALU operation, write register and memory write delay.

lw: 49 ns
sw: 43 ns
add: 37 ns
beq: 28 ns

Write register, ALU, memory write

ALU: ______

Write Register: ______

Memory Write: ______

What is the minimum clock period for the Multi Cycle processor? ______

What are possible ways to decrease the Single Cycle processor clock period taking in consideration all instructions? (Select all that apply)

1. Decrease memory write delay
2. Decrease memory read delay
3. Decrease ALU operation delay
4. Increase write register delay
5. Decrease read register delay

Independently changing which of the following component delays from the delays given above would result in the SW instruction having a longer critical path delay than LW? (Select all that apply)

1. Memory Read: 4 ns
2. Read Register: 9 ns
3. Memory Write: 31 ns
4. Memory Read: 15 ns
5. ALU: 2 ns

A program executes 10,000 instructions with the following instruction composition. Calculate the execution time of the program under the Single Cycle and Multi Cycle datapaths.

Assume the following clock periods:
Single Cycle: 141 ns
Multi Cycle: 59 ns

Opcode	**Percent of Instructions Executed**
ADD	11%
NOR	9%
LW	23%
SW	21%
BEQ	22%
NOOP	14%

Single Cycle: _______

Multi Cycle:_______

Question 6: Single Cycle/Multi Cycle Performance (Version B)

Below are the delays for each component considered in the LC2K single and multicycle datapaths discussed in lecture:

Memory Read: 22 ns
Memory Write: ?
Read Register: 10 ns
Write Register: ?
ALU: ?
All Other Operations: 0 ns

Assume the following delays for the instructions given below. Calculate the missing ALU operation, write register and memory write delay.

lw: 88 ns
sw: 102 ns
add: 66 ns
beq: 47 ns

Write register, ALU, memory write

ALU: ______

Write Register: ______

Memory Write: ______

What is the minimum clock period for the Multi Cycle processor? ______

What are possible ways to decrease the Single Cycle processor clock period taking in consideration all instructions? (Select all that apply)

1. Decrease memory write delay
2. Decrease memory read delay
3. Decrease ALU operation delay
4. Increase write register delay
5. Decrease read register delay

Independently changing which of the following component delays from the delays given above would result in the SW instruction having a longer critical path delay than LW? (Select all that apply)

1. Memory Read: 4 ns
2. Read Register: 9 ns
3. Memory Write: 40 ns
4. Memory Read: 31 ns
5. ALU: 2 ns

A different program executes 10000 instructions with the following instruction composition. Calculate the execution time of the program under the Single Cycle and Multi Cycle datapaths.

Assume the following clock periods:
Single Cycle: 156 ns
Multi Cycle: 43 ns

Opcode	**Percent of Instructions Executed**
ADD	17%
NOR	13%
LW	28%
SW	29%
BEQ	4%
NOOP	9%

Single Cycle: _______

Multi Cycle: _______

Question 6: Single Cycle/Multi Cycle/Pipeline Performance (Version C)

Below are the delays for each component considered in the LC2K single and multicycle datapaths discussed in lecture:

Memory Read: 14 ns
Memory Write: ?
Read Register: 45 ns
Write Register: ?
ALU: ?
All Other Operations: 0 ns

Assume the following delays for the instructions given below. Calculate the missing ALU operation, write register and memory write delay.

lw: 91 ns
sw: 86 ns
add: 77 ns
beq: 66 ns

Write register, ALU, memory write

ALU: ______

Write Register: ______

Memory Write: ______

What is the minimum clock period for the Multi Cycle processor? ______

What are possible ways to decrease the Single Cycle processor clock period taking in consideration all instructions? (Select all that apply)

1. Decrease ALU operation delay
2. Decrease memory write delay
3. Decrease memory read delay
4. Increase write register delay
5. Decrease read register delay

Independently changing which of the following component delays from the delays given above would result in the SW instruction having a longer critical path delay than LW? (Select all that apply)

1. Memory Read: 10 ns
2. Read Register: 9 ns
3. Memory Read: 15 ns
4. ALU: 29 ns
5. Memory Write: 31 ns

A different program executes 10000 instructions with the following instruction composition. Calculate the execution time of the program under the Single Cycle and Multi Cycle datapaths.

Assume the following clock periods:
Single Cycle: 199 ns
Multi Cycle: 45 ns

Opcode	**Percent of Instructions Executed**
ADD	36%
NOR	2%
LW	31%
SW	20%
BEQ	2%
NOOP	9%

Single Cycle: _______

Multi Cycle: ________

Question 6: Single Cycle/Multi Cycle/Pipeline Performance (Version D)

Below are the delays for each component considered in the LC2K single and multicycle datapaths discussed in lecture:

Memory Read: 21 ns
Memory Write: ?
Read Register: 24 ns
Write Register: ?
ALU: ?
All Other Operations: 0 ns

Assume the following delays for the instructions given below. Calculate the missing ALU operation, write register and memory write delay.

lw: 110 ns
sw: 127 ns
add: 89 ns
beq: 58 ns

Write register, ALU, memory write

ALU: ______

Write Register: ______

Memory Write: ______

What is the minimum clock period for the Multi Cycle processor? ______

What are possible ways to decrease the Single Cycle processor clock period taking in consideration all instructions? (Select all that apply)

1. Increase write register delay
2. Decrease read register delay
3. Decrease memory write delay
4. Decrease ALU operation delay
5. Decrease memory read delay

Independently changing which of the following component delays from the delays given above would result in the SW instruction having a longer critical path delay than LW? (Select all that apply)

1. Memory Read: 28 ns
2. Read Register: 9 ns
3. Memory Read: 35 ns
4. ALU: 2 ns
5. Memory Write: 50 ns

A different program executes 10000 instructions with the following instruction composition. Calculate the execution time of the program under the Single Cycle and Multi Cycle datapaths.

Assume the following clock periods:
Single Cycle: 123 ns
Multi Cycle: 61 ns

Opcode	**Percent of Instructions Executed**
ADD	18%
NOR	9%
LW	24%
SW	10%
BEQ	32%
NOOP	7%

Single Cycle: _______

Multi Cycle: _______

Question 6: Single Cycle/Multi Cycle/Pipeline Performance (Version E)

Below are the delays for each component considered in the LC2K single and multicycle datapaths discussed in lecture:

Memory Read: 23 ns
Memory Write: ?
Read Register: 26 ns
Write Register: ?
ALU: ?
All Other Operations: 0 ns

Assume the following delays for the instructions given below. Calculate the missing ALU operation, write register and memory write delay.

lw: 119 ns
sw: 118 ns
add: 96 ns
beq: 66 ns

Write register, ALU, memory write

ALU: ______

Write Register: ______

Memory Write: ______

What is the minimum clock period for the Multi Cycle processor? ______

What are possible ways to decrease the Single Cycle processor clock period taking in consideration all instructions? (Select all that apply)

1. Decrease memory read delay
2. Decrease ALU operation delay
3. Decrease read register delay
4. Increase write register delay
5. Decrease memory write delay

Independently changing which of the following component delays from the delays given above would result in the SW instruction having a longer critical path delay than LW? (Select all that apply)

1. Read Register: 43 ns
2. Memory Write: 61 ns
3. Memory Read: 14 ns
4. Memory Read: 20 ns
5. ALU: 22 ns

A different program executes 10000 instructions with the following instruction composition. Calculate the execution time of the program under the Single Cycle and Multi Cycle datapaths.

Assume the following clock periods:
Single Cycle: 189 ns
Multi Cycle: 33 ns

Opcode	**Percent of Instructions Executed**
ADD	32%
NOR	9%
LW	13%
SW	32%
BEQ	3%
NOOP	11%

Single Cycle: _______

Multi Cycle: _______

Question 7: New ISA

To improve on LC2K, we've developed a new, modern ISA called the LCJINX with 12 opcodes and 16 registers. Unfortunately, since our design team got lazy, we can still only support 16 bit signed immediate values. (note: Jinx definitely did not write this question)

There are four types of instructions:

- J-type instructions take in three operands, which are all registers.
- I-type instructions take in three operands, where two are registers and the last is an immediate value.
- N-type instructions take in no operands.
- X-type instructions take in four operands, where all four operands are registers.

1) **What is the minimum word size of the LCJINX? <u>Round your answer up to the nearest byte.</u> (3 points)**

Assume the LCJINX is word-addressable, with a word size of 8 bytes. The instructions for the LCJINX are as follows:

Instruction	**Type**	**Opcode**	**Description**
add rA rB rD	J	0x0	Adds the values in registers A and B and stores the result to register D (destination).
nand rA rB rD	J	0x1	Takes the bitwise NAND of the values in registers A and B and stores the result to register D (destination).
addi rA, rD, imm	I	0x2	Adds the value in register A with the immediate value and stores the result to register D (destination).
sft rA, rB, imm	I	0x3	Takes the value in register A and shifts it by the number of bits specified by *imm*. If *imm* is positive, then the value is shifted to the right by *imm* bits. If *imm* is negative, the value is shifted to the left by |*imm*| bits. The result is stored into register B.
ld rA, rB, imm	I	0x4	Loads the value in memory at address (value of register A + immediate value) to register B.
str rA, rB, imm	I	0x5	Stores the value in register B to memory address (value of register A + immediate value).
beq rA, rB, imm	I	0x6	If the values in rA and rB are the same, then branch to PC = *imm*. (note: different from LC2K's beq!)
blt rA, rB, imm	I	0x7	No, not the sandwich. If the value in register A is less than (<) the value in register B, then branch to PC = *imm*.
meow	N	0x8	Equivalent to LC2K's *noop* instruction. Does nothing.
halt	N	0x9	Equivalent to LC2K's *halt* instruction. Halts the program.
cmov rA, rB, rC, rD	X	0xA	If the values of register A and register B are equal, set the value of register C to be the value in register D.
beqlr rA, rB, rC, rD	X	0xB	If the values of register A and register B are equal, store the current PC+1 into register D and branch to the value in register C. This is similar to LC2K's *jalr* instruction.

For backwards compatibility and ease of use, the *.fill* directive works exactly the same as LC2K. Labels have a maximum length of 12 characters, for the optimal programmer experience (or as much as you can get while writing in assembly). All symbolic addresses resolve to the memory address of that label. Assume all registers start with the value 0.

J-type instructions are encoded as follows:

- **bits 35-32: opcode**
- **bits 31-28: register A**
- **bits 27-24: register B**
- **bits 23-4: unused (all 0)**
- **bits 3-0: register D**

I-type instructions are encoded as follows:

- **bits 35-32: opcode**
- **bits 31-28: register A**
- **bits 27-24: register B**
- **bits 23-16: unused (all 0)**
- **bits 15-0: 16-bit immediate**

N-type instructions are encoded as follows:

- **bits 35-32: opcode**
- **bits 31-0: unused (all 0)**

X-type instructions are encoded as follows:

- **bits 35-32: opcode**
- **bits 31-28: register A**
- **bits 27-24: register B**
- **bits 23-20: register C**
- **bits 19-16: register D**
- **bits 15-0: unused (all 0)**

2) Complete the machine code translation of the LCJINX code shown below. Express your answers in hex.
Assuming all registers start as 0 at the beginning of the program, what is the value in register 5? (7 points, 4.5 for MC and 2.5 for register)

LCJ1NX Assembly	**Machine Code**
`addi  1     1     16`	`0x211000010`
`sft   1     2     3`	`0x312000003`
`nand  0     0     3`	`0x100000003`
`add   2     3     4`	`0x_________`
`sft   4     4     -4`	`0x_________`
`cmov  1     4     5     3`	`0x_________`
`halt`	`0x900000000`

3) **Your team has decided to use the 370-developed J1 chip to run LCJINX, and you're now tasked with a program to run binary search in a sorted array. Your teammates had a working prototype, but a sleep- and caffeine-deprived programmer accidentally deleted a few lines of code. Help restore the program below based on the comments. (10 points)**

```
           ld    0    14   searchAddr
           beqlr 0    0    14   15    // call binary search function
           halt
           meow
searchFunc add   0    0    1          // l = 0
           ld    0    2    arraySize  // r = array size
           ld    0    9    target     // load target
loop       beq   1    2    nofind     // if l == r, not found
           ____  _    _    ______     // find m = (l+r) / 2 (2 lines)

           ____  _    _    ______     // store m into reg 3

           ____  _    _    ______     // load array[m] into reg 5

           beq   _    _    ______          // if array[m] == target

           ____  _    _    ______     // handle inequalities

           ____  _    _    ______     // else

           ____  _    _    ______     // else
equal      add   0    3    13         // return m in reg 13
           beq   0    0    done       // done!

______     ____  _    _    ______     // inequality case 1

           ____  _    _    ______          // inequality case 1

done       beqlr 0    0    15   14    // return

nofind     ____  _    _    ______     // load return value with -1
           beqlr 0    0    15   14    // return
searchAddr .fill searchFunc
arraySize  .fill 8
target     .fill 7
arrayAddr  .fill array
array      .fill 1
           .fill 3
           .fill 4
           .fill 7
           .fill 10
           .fill 12
           .fill 13
           .fill 16
```

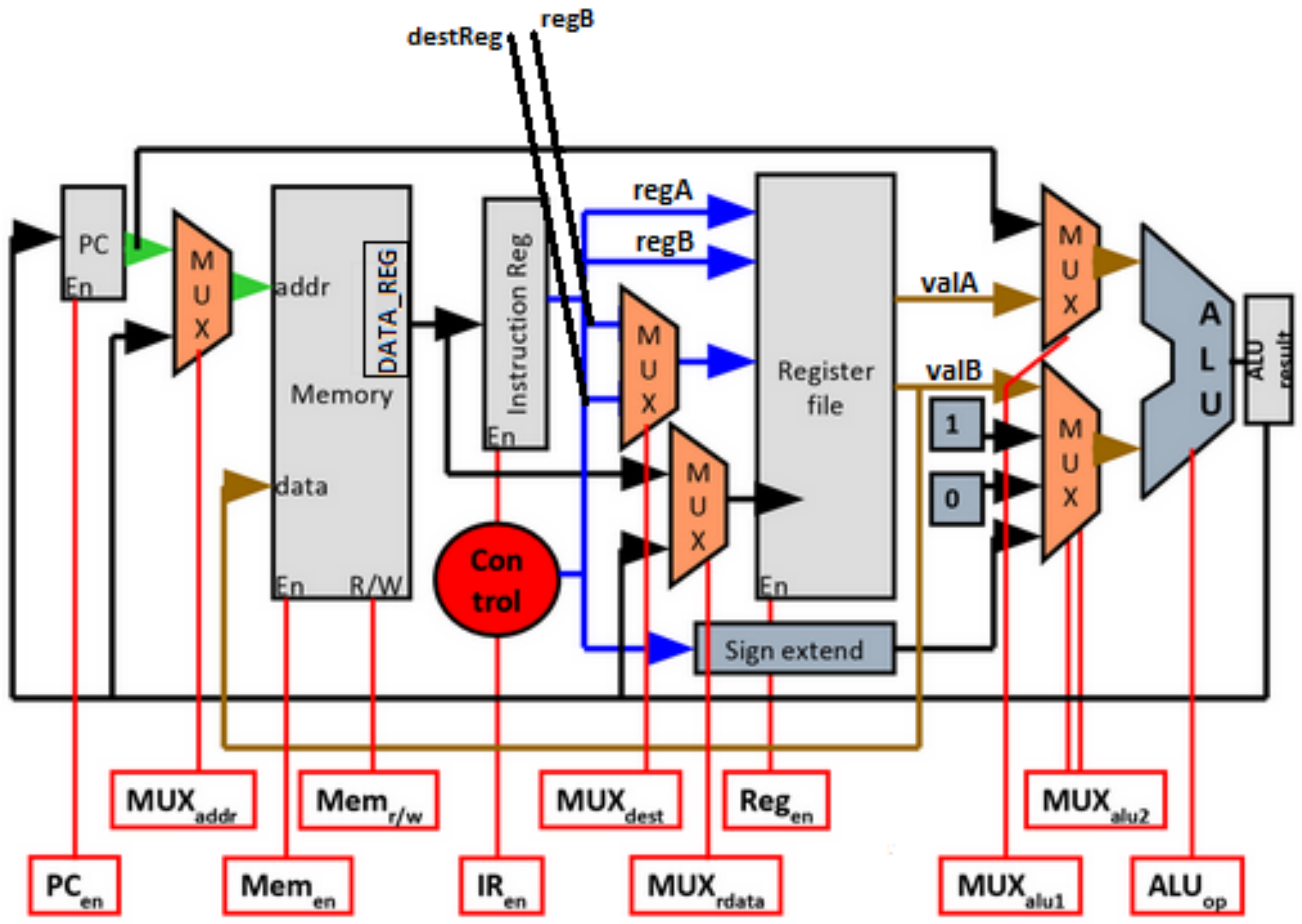
Question 8: Multi-cycle Datapath Design Points: ___/20

Consider a new LC2K I-type instruction:

```
accmems    regA  regB  stride
```

The goal of this instruction is to facilitate strided accumulation of the data in the memory. Execution semantics of this instruction is as follows:

```
regB = regB + Mem[regA];
regA = regA + stride;
```



a) You have been asked to change the LC2K multi-cycle datapath to support the new instruction. We can **only modify the MUXes** (and add any necessary connections to them). Note that none of the MUXes should have more than four inputs to them. Describe your changes to datapath below. (`accmems` **should take 5 cycles or less** to complete) (5pts)

Example:

MUX_example1:	MUX_example2:
DATA_REG added to input 10.	NA // Leave blank or type NA if unchanged.

- MUX_addr: ______________________ MUX_dest: ______________________
- MUX_rdata: ______________________ MUX_alu1: ______________________
- MUX_alu2 ______________________

b) Implement `accmems` in the multi-cycle data-path in five or fewer cycles. Specify the operations for each cycle. Provide an informal description (few words) followed by exact changes to the multicycle state. Assume [data_reg] stores the value read from memory. (10 pts) *You may not need to fill all the blanks.*

Cycle 1: Fetch
 [Instruction_Reg] = Mem[PC]
 [ALU_Result] = PC + 1
Cycle 2: Decode
 [PC] = [ALU_Result]
 Read register values
Cycle 3:
 ________________ = ______________________
 ________________ = ______________________
Cycle 4:
 ________________ = ______________________
 ________________ = ______________________
Cycle5:
 ________________ = ______________________
 ________________ = ______________________

c) Write the control signals for all the cycles needed for the "accmems" instruction on the modified datapath after your connections have been added. (5 pts)
You can assume the following things:

- All connections added to any mux are added at the bottom of the mux
- The select bits for extended muxes may need to be increased from the original mux

PC (en)	**MUX (addr)**	**MEM (en)**	**MEM (r/w)**	**IR (en)**	**MUX (dest)**	**MUX (rdata)**	**Reg (write en)**	**MUX (alu1)**	**MUX (alu2)**	**ALU (op)**
0	0	1	0	1	X	X	0	0	01	0
1	X	0	X	0	X	X	0	X	XX	X

Question 9: C to LC2K

This question was removed from the final version of the exam, but has been available in practice exams since.

1. Convert C code to LC2K by filling the blanks.
   Note this does not follow the standard LC2K ABI:
   reg2 has the start address of arr.
   reg4 is the input `real` of `find_number`.
   reg5 is the stack pointer.

reg3: size.
reg6: -1
reg4: 4 (real).

C	LC2K
`struct number{`	`main.as`
`    int real;`	`    lw    0    2    arr`
`    int imagine;`	`    lw    0    3    size`
`}`	`    lw    0    6    neg1`
	`    lw    0    4    real`
`struct number arr[5];`	`    add   3    6    3` → size: size-1
	`    lw    0    6    faddr`
`extern int find_number(struct`	`    jalr  6    1`
`number arr[], int idx, int real)` ↑reg4	`    halt`
	`arr    .fill   Array`
`int main(){`	`size   .fill   5`
`    find_number(arr, 4, 3);` ↑	`neg1   .fill  -1`
`}`	`real   .fill   3`
	`Array  .fill   3`
	`       .fill   3`
	`       .fill   2`
	`       .fill   3`
	`       .fill   3`
	`       .fill   5`
	`       .fill   8`
	`       .fill   3`
	`       .fill   5`
	`       .fill   0`

C	LC2K

`struct number{`
      `int real;` 4
      `int imagine;` 4   7 = idx × 8
`}` 8
 	
`int find_number(struct number arr[], int idx,`
`int real){` 2
    `if (arr[idx].real == real){` 4 ↑
        `return idx;`
    `}`
    `else if(idx == 0){`
        `return -1;`
    `}`
    `else{`
        `return find_number(arr, idx-1, real)`
    `}`
`}`
	`Func add _ _ 7 //step1: arr[idx]`
	`add 7 2 7 //step2: arr[idx]`
	`lw 7 7 0 //step3: arr[idx]`
	`=`
	`beq 7 4 ret`
	
	`beq 3 0 ______`
	`lw 0 6 neg1`
	`add 6 3 3`
	
	`lw 0 6 ______`
	`lw 0 7 one`
	`sw 5 1 Stack`
	`add 7 5 5`
	
	`jalr _ _`
	`lw 0 7 neg1`
	`add 7 5 5`
	`lw 5 1 Stack`
	
	`beq 0 0 ______`
	`noHit lw 0 3 neg1`
	
	`ret jalr _ _`
	`Faddr .fill Func`
	`neg1 .fill -1`
	`one .fill 1`

2. What is register 3 used for? (select all apply)

A. return address
B. idx
C. arr
D. real
E. Stack pointer
F. return_value
G. temporary register

3. Which register has the return address? Is this register a caller save register or a callee save register? (select all apply)

A. r1
B. r2
C. r3
D. r4
E. r5
F. r6
G. r7
H. Caller save register
I. Callee save register

4. What is the value that `find_number` returns?

2